PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

EFFICIENT UNIFICATION WITH INFINITE TERMS IN LOGIC PROGRAMMING

Alberto Martelli and Gianfranco Rossi

Dipartimento di Informatica
Universita' di Torino
C.so M. D'Azeglio, 42
10125 TORINO - ITALY

ABSTRACT

Proposals have been recently put forward to extend logic programming so as to deal with infinite terms. In this paper we present an algorithm for unification with infinite terms and we show that it is simpler and more efficient than previously proposed algorithms both for infinite terms and for the standard finite case. Finally we point out that interpreters for logic programming languages allowing infinite terms can be substantially more efficient than standard interpreters.

1 INTRODUCTION

Unification algorithms play a fundamental role in the implementation of programming languages based on logic (such as Prolog [Clocksin and Mellish 1981]) or on term rewriting. In fact, unification must be performed at every execution step, and thus the efficiency of this operation affects crucially the efficiency of the whole interpreter.

In the past, many efforts were devoted to developing efficient unification algorithms [Huet 1976] [Paterson and Wegman 1978] [Martelli and Montanari 1982] [Corbin and Bidoit 1983], and many studies are under way for devising parallel algorithms suitable for hardware implementation.

However, in the implementation of a logic programming language, the performance of a unification algorithm must be assessed not for a single step but over the entire computation and even the most efficient algorithm can be considered intolerably slow. For instance, the "occur check" that must be performed by a unification algorithm to make sure that no variable is bound to a term containing it, is suppressed in many implementations of Prolog interpreters being considered too expensive. The consequence is that such an interpreter is incorrect, since it is not able to detect cyclic definitions such as, for instance, the one generated by the following program

```
p(X,X).
?- p(X,f(X)).
```

Recently, Colmerauer [Colmerauer 1982] has proposed a novel theoretical model of logic programming languages involving infinite trees, and has presented convincing examples of the usefulness of such semantics. This model requires a unification algorithm which does not perform the "occur check", but the above mentioned Prolog interpreters are not correct in this case too, since their unification algorithm might loop in the unification of infinite terms, as can be checked by running the following Prolog program

```
p(X,X).
?- p(X,f(X)), p(Y,f(Y)), p(X,Y).
```

In the same paper Colmerauer presents also a correct (terminating) algorithm for unification with infinite terms. The problem of unifying infinite terms was studied in Huet's thesis [Huet 1976] where the existence of a most general unifier is proved for finite and infinite terms, and an algorithm to compute it is given. More recent papers dealing with this topic are [Mukai 1983] [Fages 1983a] [Fages 1983b] [Filgueiras 1984] [Haridi and Sahlin 1984].

In this paper we propose an algorithm that is simpler and more efficient than previously proposed algorithms. We show also that well known algorithms for the finite case (among them the algorithm of [Martelli and Montanari 1982]) can be obtained from it by adding suitable operations for detecting cycles, thus proving that unification algorithms for infinite terms may be simpler than those for

finite terms.

Finally we analyze the use of these algorithms in an interpreter for logic programming languages, showing that an interpreter for the infinite case can be substantially more efficient than an interpreter for standard first-order logic.

2 BASIC DEFINITIONS

Given two terms t' and t", the standard unification problem can be written as an equation t' = t". A solution of the equation, called a *unifier*, is any substitution, if it exists, which makes the two terms identical. A *substitution* is a set of ordered pairs

$$\{<x_1,t_1>,\ldots,<x_n,t_n>\},$$

where the x_i's are distinct variables and the t_i's are terms, such that $x_i \neq t_i$. For istance, a unifier of the equation

$$f(h(x),y,z,g(z))=f(h(g(y)),z,a,w)$$

is

$$\{<x,g(y)>,<y,z>,<z,a>,<w,g(z)>\}.$$

It is convenient to consider also *systems of equations*, that is finite sets of equations. Again, a unifier is any substitution which makes the pairs of terms of all equations identical simultaneously. A system of equations of the form

$$\{x_1=t_1,\ldots,x_n=t_n\},$$

where the x_i's are distinct variables, is said to be in *solved form*. It is easy to see that such a system has always an immediate solution. For instance, a solution of

$$\{x=f(a,y),z=h(w),y=g(z,b)\}$$

is

$$\{<x,f(a,g(h(w),b))>,\ <z,h(w)>,\ <y,g(h(w),b)>\}$$

which is obtained by repeatedly substituting variables with their corresponding right-hand sides.

A system in solved form can contain cycles, namely, can contain a variable which is directly or indirectly bound to a term containing it. In this case the solution of the system contains *infinite terms*. For instance, a unifier of

$$\{x=f(x,y),\ y=a\}$$

is

$$\{<x,f(f(f(\ldots),a),a),a)>,<y,a>\}$$

where x is bound to an infinite term. The close correspondence between systems in solved form and unifiers, allows us to use the more convenient notation of a system in solved form in place of the corresponding unifier, which may contain infinite terms.

To describe the algorithms of this paper, we require some other definitions. We will use the terminology introduced in [Martelli and Montanari 1982].

A *multiequation* S = M, where S is a nonempty set of variables and M is a multiset of nonvariable terms, allows us to group together many equations. For instance, the system of equations

$$\{t'=t",x=t',x=y\}$$

can be easily transformed into the equivalent multiequation:

$$\{\{x,y\}=(t',t")\}.$$

A *system of multiequations* is a finite set of multiequations $S_i=M_i$, $i=1,\ldots,n$. It is in *solved form* if all left-hand sides S_i are disjoint and all right-hand sides M_i consist of no more than one term.

The *common part* of a multiset M of terms (variables or not) is a term which, intuitively, is obtained by superimposing all terms of M and by taking the part which is common to all of them starting from the root. For instance, given the multiset of terms

$$(f(x,g(h(a),v),y),f(h(y),g(x,b),z))$$

its common part is

$$C:\ f(x,g(x,v),y).$$

The *frontier* is a set of multiequations, where every multiequation is associated with a leaf of the common part and consists of all subterms (one for each term of M) corresponding to that leaf. The frontier of the above multiset of terms is

$$F:\ \{\{x\}=(h(y),h(a)),\ \{v\}=b,\ \{y,z\}=\emptyset\}.$$

Note that if there is a clash of function symbols among some terms, then M has no common part and frontier. In this case the terms of M are not unifiable.

More rigorous definitions and proofs of the above concepts are given in [Martelli and Montanari 1982].

3 UNIFICATION WITH INFINITE TERMS

A unification algorithm starts with a system of multiequations and repeatedly applies transformations until a system in solved form is obtained. Transformations are such that they produce equivalent systems, that is they preserve the sets of all unifiers. We introduce the following two transformations.

COMPACTION. Given a system R containing two multiequations S=M and S'=M', with $S \cap S' \neq \emptyset$, the new system R' is obtained by replacing the two multiequations with $S \cup S' = M \cup M'$.

REDUCTION. Given a system R containing a multiequation S=M, such that M is nonempty and has a common part C and a frontier F, the new system R' is obtained by replacing S=M with the multiequation S=(C) and with all the multiequations of F.

In both cases, a unifier of R is also a unifier of R' and vice-versa.

We can now give the following general non-deterministic algorithm for unification.

ALGORITHM UNIFY-0

Given a system of multiequations R, repeatedly perform any of the following actions. If neither applies, then stop with success.

(a) if there are two multiequations S=M and S'=M' with $S \cap S' \neq \emptyset$
then apply COMPACTION;
(b) if there is a multiequation S=M with size(M)>1
then compute the common part and frontier of M;
if M has no common part
then stop with failure
else apply REDUCTION.

size(M) is a function which returns the number of terms in the multiset M.

It is easy to see that the above algorithm always terminates, both in the finite and infinite case, with the final system in solved form (in the case of success). To prove termination one must consider the following remarks. With REDUCTION some function symbol is eliminated; at each confrontation of two or more function symbols only one of them is retained in the common part, while the others are discarded. This operation also introduces new variables occurrences in the left-hand sides of multiequations, but they will be eliminated by means of COMPACTIONs.

As an example, let us consider the following system of multiequations:

$$\{\{x\}=f(h(y),z) \quad \{y\}=f(z,h(x)) \quad \{x,y\}=\emptyset\ \}. \tag{1}$$

By compacting the first with the third multiequation and then the resulting one with the second, we get

$$\{\{x,y\}=(f(h(y),z),f(z,h(x))\ \}.$$

By applying transformation (b) we get

$$\{\{x,y\}=f(z,z) \quad \{z\}=h(y) \quad \{z\}=h(x)\ \}.$$

We now apply transformation (a) to the second and third multiequation, and then transformation (b) to the resulting one

$$\{\{x,y\}=f(z,z) \quad \{z\}=h(y) \quad \{x,y\}=\emptyset\ \}$$

Finally, by compacting the first and third multiequations we get the solved system

$$\{\{x,y\}=f(z,z) \quad \{z\}=h(y)\}.$$

In order to allow efficient implementations, this algorithm can be modified by assuming the system R to be divided into two systems of multiequations, R1 and R2, with the only restriction that R1 must have disjoint left-hand sides.

ALGORITHM UNIFY-1.

Repeatedly perform any of the following actions. If neither applies then stop with success.

(a) extract a multiequation from R2, and perform all possible (including none) COMPACTIONs in R1 due to the above multiequation;
(b) if there is a multiequation S=M in R1 with size(M)>1
then compute the common part and

frontier of M;
<u>if</u> M has no common part
<u>then</u> stop with failure
<u>else</u> apply REDUCTION in R1, and move the multiequations of the frontier into R2.

When the algorithm stops with success, R2 is empty and R1 holds the system in solved form.

Algorithm UNIFY-1 is general enough to allow several concrete algorithms to be easily derived from it.

For instance, we may obtain a simple deterministic algorithm by stating that action (b) must follow action (a) and (b) reduces the multiequation computed in (a), that they must be repeatedly executed until R2 is empty, and that R2 must be treated as a list. As an example, let us consider the same system as discussed earlier

R1: {{x}=f(h(y),z)
{y}=f(z,h(x) };
R2: ({x,y}=∅).

After the first iteration we have

R1: {{x,y}=f(z,z)};
R2: ({z}=h(y),{z}=h(x)).

Performing two more iterations we get

R1: {{x,y}=f(z,z)
{z}=h(y)};
R2: ({x,y}=∅).

Finally, by compacting the only multiequation of R2 with the first of R1, R2 becomes empty and we get the solved system in R1.

Note that, because of the ordering of transformations, system R1 is in solved form after each iteration. Thus, this modified version of Algorithm UNIFY-1 closely corresponds to a very common way of describing the unification process, where R2 is a set of pairs (or n-tuples) of terms to be unified and R1 represents the binding environment.

The computation of the common part and of the frontier in Algorithm UNIFY-1 could be substantially simplified by introducing intermediate variables so that every term contains no more than one function symbol (i.e. it is of the form $f(x_1,\ldots,x_n)$). In this case, the common part of a multiset of terms may be obtained by selecting anyone of them, without having to construct any new term, and all multiequations of the frontier contain only variables. With these modifications, our algorithm becomes similar to the one known in the literature as Huet's algorithm.

4 SOME REMARKS ON COMPLEXITY

Complexity analysis of our algorithm can be performed with techniques similar to those used for instance in [Martelli and Montanari 1982]. It turns out that the algorithm complexity is equal to the sum of two terms, one linear with the number of function symbols and another one almost linear with the number m of variable occurrences in the initial system. The non-linearity factor depends on the way multiequations merging is implemented and is negligible in practice. Indeed we might represent sets of variables as trees and use the well-known UNION-FIND algorithms [Aho et al.1974] to add elements and to access them with a complexity of the order of mG(m), where G is an extremely slowly growing function (the inverse of the Ackermann function).

The main feature which contributes to the linearity of our algorithm is the substitution of the right-hand side terms of a multiequation with their common part during REDUCTION. In fact, reduction of two terms assures that the two pieces of them which are compared will be replaced by just one new term, the common part, whose size (i.e. the number of variables and function occurrences in it) is surely equal or smaller than the sizes of both discarded pieces.

This feature is also the main difference among our algorithm and other algorithms dealing with infinite terms, such as those in [Colmerauer 1982] or in [Mukai 1983].

Let us consider Colmerauer's algorithm in more details, through a simple example and using our terminology. Given the system

{{x}=f(g(x))
{y}=f(z)
{x,y}=∅}

it can be transformed into

{{y}=f(g(y))
{y}=f(z)
{x,y}=∅}

by applying a transformation called

"Variable Elimination", whose result is similar to multiequations merging (it may be implemented in a similar way, too). Then the first two multiequations can be "reduced" by applying another transformation, called "Confrontation"

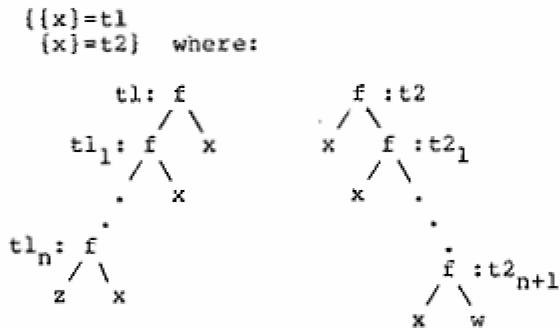
$$\{\{y\}=f(z)$$
$$f(z)=f(g(y))$$
$$\{x,y\}=\emptyset\}.$$

The multiequation with the smallest size term has been retained, while the other has been discarded (this guarantees algorithm's termination). Finally, the frontier of two terms can be computed by repeatedly applying the "Splitting" transformation

$$\{\{y\}=f(z)$$
$$\{z\}=g(y)$$
$$\{x,y\}=\emptyset\}.$$

We can conclude that the two algorithms are very similar, except for reduction of terms. Colmerauer's algorithm does not create any new term; it performs reduction using only existing terms. Moreover it requires an explicit test for termination based on the comparison of the size of the terms which are reduced.

As a consequence this algorithm may be no longer linear in the number of function symbols, as clearly depicted by the following example:



By applying Confrontation and Splitting we get

$$\{\{x\}=t1,\{x\}=t1_1,\{x\}=t2_1\}.$$

By applying Confrontation to the first two multiequations we get

$$\{\{x\}=t1_1,t1=t1_1,\{x\}=t2_1\},$$

where x is bound to the smaller term. Splitting of $t1=t1_1$ requires a number of comparisons of the order of n, and the result is

$$\{\{x\}=t1_1,\{x\}=t2_1,\{z\}=f(z,x)\}.$$

This system is similar to the initial one, with $t1_1$ substituted for t1 and $t2_1$ for t2. We can thus repeatedly apply the same sequence of transformations as above, having at each step to solve the equations $t1_1=t1_2$, $t1_2=t1_3$,... . As pointed out before, splitting of each of these equations requires a time proportional to the depth of terms, therefore summing up to a quadratic execution time. Further complexity is added to this algorithm by the task of selecting the smallest term.

On the contrary, using Algorithm UNIFY-1, at each step we go down the trees just one level and each function symbol is encountered only once.

5 UNIFICATION WITH FINITE TERMS

Algorithm UNIFY-1 can easily be extended to deal with the standard unification problem with finite solutions, by checking, after termination, that the R1 part of the system does not contain any cycle. Checking can be done with a topological sorting algorithm, which does not increase substantially the complexity of the unification algorithm.

Topological sorting can be embedded in the unification algorithm by adding counters to multiequations, as suggested in [Martelli and Montanari 1982]. Since multiequations in R1 have disjoint left-hand sides, a counter can be associated with every multiequation of R1, counting the number of other occurrences in R1 and in R2 of the variables in the left-hand side of the associated multiequation. For instance:

```
R1: {{x}=f(h(v),z,h(w))   [1]
     {y}=f(z,h(a),z)      [1]
     {z}=∅                [3]   (2)
     {v}=∅                [1]
     {w}=∅ };             [1]
R2: {{x,y}=∅}.
```

Note that, using counters requires that system R1 has a multiequation for each different variable it contains, even if the variable has no binding.

Counters are updated at every step of the algorithm. Whenever a multiequation is extracted from R2 in the COMPACTION phase (step (a)), the associated counters are decremented by one, and if two or more multiequations are merged together, their counters

are summed up. Instead, a counter is incremented whenever an occurrence of the associated variables appears in the common part computed in the REDUCTION phase (step (b)).

For instance, after the execution of step (a) and step (b) of the modified algoritnm, counters of system (2) are updated as follows:

```
R1: {{x,y}=f(z,z,z)        [0]
     {z}=Ø                 [6]
     {v}=Ø                 [1]
     {w}=Ø }               [1]
R2: {{z}=h(v),{z}=h(a),{z}=h(w)}.
```

A new action must be added to Algoritnm UNIFY-1 in order to detect variables which do not occur elsewhere and to mark them as not being part of any cycle, properly updating counters:

(c) <u>if</u> there is a multiequation S=M in R1 with size(M)=1 and with counter equal to zero and not yet marked
<u>then</u> for each occurrence of the variables in M, decrease the corresponding counters and mark the multiequation.

For instance, applying this action to the first multiequation in the above system, causes the counter associated with {z} to be decremented by 3 and the multiequation to be marked (e.g. with a star)

```
R1: {{x,y}=f(z,z,z)        [0]*
     {z}=Ø                 [3]
     ...}.
```

After few more iterations, we get the final solved system

```
R1: {{x,y}=f(z,z,z)        [0]*
     {z}=h(v)              [0]*
     {v,w}=a               [0]*
R2: Ø
```

which does not contain any cycle since all its counters have value zero, and they are all marked.

As pointed out in the previous section, several algorithms can be derived from the above nondeterministic algorithm, by specifying the order in which to apply transformations.

In particular, if REDUCTION is delayed until the counter of a multiequation goes to zero, we obtain the algorithm of [Martelli and Montanari 1982] whose main feature is the capability of detecting cycles earlier.

For instance, let us consider a slightly modified version of system (2)

```
R1: {{x}=f(h(v),z,h(w))    [2]
     {y}=f(z,h(g(x)),z)    [1]
     {z}=Ø                 [3]
     {v}=Ø                 [1]
     {w}=Ø };              [1]
R2: {{x,y}=Ø}.
```

It is easy to see that this system contains a cycle. If no restriction is imposed on the applicability of REDUCTION, the above algorithm with counters can detect the cycle only when the final solved system is obtained, looking at its counters

```
R1: {{x,y}=f(z,z,z)        [1]
     {z}=(h(v))            [3]
     {v,w}=g(x)            [1]
R2: Ø.
```

On the contrary, if REDUCTIONs are delayed like in [Martelli and Montanari 1982], after few steps we get the system

```
R1: {{x,y}=f(z,z,z)                  [1]
     {z}=(h(v),h(g(x)),h(w))         [3]
     {v}=Ø                           [1]
     {w}=Ø};                         [1]
R2: Ø.
```

At this point, REDUCTION over {z} is not possible because its associated counter is not zero. No other action is possible; thus the algorithm can stop and detect a cycle.

6 UNIFICATION IN LOGIC PROGRAMMING INTERPRETERS

Algorithms of the previous sections for the finite and infinite case have comparable complexity as long as they are used to solve a single unification problem. The great difference between them becomes apparent when they are used in successive steps in the implementation of a logic programming language interpreter.

Let us define more precisely the meaning of logic program computation.

Following [Colmerauer 1982], a <u>logic program</u> is defined as a set of rules of the form:

$$t \rightarrow t_1 \ldots t_n$$

where t, t_1, ..., t_n are terms and n can be 0.

A computation <u>state</u> is a pair (S,E) where S is a sequence of terms $s_0, s_1, \ldots, s_m$, $m>0$, and E is a system

of multiequations.

A transition from a state (S,E) to a new state (S',E'), by means of a logic program P, is defined as follows:

$$(s_0\ s_1 \ldots s_n, E) \Rightarrow (t_1 \ldots t_n\ s_1 \ldots s_m, E \cup \{\{x\}=(s_0,t)\})$$

iff there is a rule $t \to t_1 \ldots t_n$, belonging to P, such that the system of multiequations $E \cup \{\{x\}=(s_0,t)\}$ has a solution. To avoid name conflicts, variables in the selected rule must be appropriately renamed and x must be a new variable neither contained in E nor in (s_0,t).

Finally, given an initial state (S_0,E_0), a computation of P may be defined operationally as a sequence of state transitions

$$(S_0,E_0) \Rightarrow (S_1,E_1) \Rightarrow \ldots \Rightarrow (\emptyset,E_n).$$

The result of the computation is the solution of the final system E_n. Since a transition can be applied only if the resulting system has a solution, we can assume all systems E_i to be in solved form. Then the solution of E_n is computed incrementally by taking, at each step i, a solved system E_{i-1}, adding a new multiequation, and solving the new system.

It is easy to see that such an incremental solution process is equivalent, as far as complexity is concerned, to solve the system only at the last step of the computation, since it is always possible to apply the same sequence of transformations in both cases. Therefore the complexity of the computation is (almost) linear with the number of function symbols of all terms of the rules involved in the computation.

This is no longer true if only finite terms are allowed, since checking the lack of cycles at every step is more expensive than doing it once at the end of the computation. For instance, as pointed out in [Colmerauer 1982], the "occur check" forces a program for appending two lists, to visit its first argument at every step, thus taking a quadratic time, even if the interpreter uses a linear unification algorithm. No known algorithm can do the "occur check" incrementally in an efficient way. Improvements in efficiency are possible by performing some static analysis of programs for detecting at compile time places where loops can occur, as described for instance in [Plaisted 1984].

7 CONCLUSIONS

We have presented a simple abstract algorithm for unification with infinite terms, and we have shown that very efficient concrete algorithms (with almost linear complexity) can be derived from it both for infinite and finite terms. We have also shown that the algorithm proposed by Colmerauer for infinite terms is less efficient than ours, by giving an example where that algorithm takes a quadratic execution time.

Finally we have pointed out that interpreters for logic programming languages allowing infinite terms can be much more efficient than interpreters for languages with finite terms only.

The proposed algorithm can be used to write correct interpreters whose efficiency is comparable to that of the incorrect interpreters which use standard unification algorithms without "occur check". Of course, memory management and structure sharing problems must be dealt with using tecniques similar to those used to implement Prolog interpreters [Bruynooghe 1982] [Mellish 1982]. Our next efforts will be devoted to these topics.

REFERENCES

Aho, A.V., Hopcroft, J.E. and Ullman, J.D. The Design and Analysis of Computer Algorithms; Addison-Wesley, New York, 1974.

Bruynooghe, M. The memory management of PROLOG implementations; in Logic Programming (K.L. Clark and S-A. Tarnlund eds.), Academic Press, 1982, 83-98.

Clocksin, W.F. and Mellish, C.S. Programming in Prolog; Springer-Verlag, 1981.

Colmerauer, A. Prolog and Infinite Trees; Logic Programming (K.L. Clark and S-A. Tarnlund eds.), Academic Press, 1982.

Corbin, J. and Bidoit, M. A Rehabilitation of Robinson's Unification Algorithm; in Proc. of the IFIP 9th Congress, Paris, France, Sept. 19-23,

1983.

Fages, F. Note sur l'unification des termes de premier ordre finis et infinis; Rapport LITP 83-29, May 1983.

Fages, F. Formes canoniques dans les algèbres booléennes et applications à la demonstration automatique; Thèse de 3ème cycle, Université Paris VI, June 1983.

Filgueiras, M. A Prolog interpreter working with infinite terms; in implementations of PROLOG (J.A. Campbell ed.), J.Wiley and Sons, 1984, 250-258.

Haridi, S. and Sahlin, D. Efficient implementation of unification of cyclic structure; implementations of PROLOG (J.A. Campbell ed.), J. Wiley and Sons, 1984, 234-249.

Huet, G. Résolution d'équations dans les languages d'ordre 1,2,...,ω.; These d'etat, Universite' Paris VII, 1976.

Martelli, A. and Montanari, U. An Efficient Unification Algorithm; ACM TOPLAS, 4, 2 (April 1982).

Mellish, C.S. An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter; in Logic Programming (K.L. Clark S-A. Tarnlund eds.), Academic Press, 1982, 99-106.

Mukai, K. A Unification Algorithm for Infinite Trees; in Proc. of the 8th Int. Conf. on Artificial Intelligence; Karlsruhe, W. Germany, 8-12 August 1983.

Paterson, M.S. and Wegman, M.N. Linear Unification; J. Comput. Syst. Sci., 16, 2 (April 1978).

Plaisted, D.A. The occur-check problem in Prolog; in Proc. of the 1984 Int. Symp. on Logic Programming, Atlantic City, N.J., Feb. 6-9,1984.